目錄

介紹 1.1
概念 1.2
資料庫 1.3
- 資料排列組合 1.3.1
- MySQL 1.3.2
  - 資料類型 1.3.2.1
  - 程式語法 1.3.2.2
  - 索引 1.3.2.3
  - 鎖表 1.3.2.4
  - 交易 1.3.2.5
  - 伺服器 1.3.2.6
  - SELECT 語法效能測試 1.3.2.7

程式 1.4
- 陣列 1.4.1

快取 1.5
- 設定 1.5.1
- 刪除 1.5.2

檔案 1.6
網站架構演進 1.7
- Web 與 Database 在一起 1.7.1
- Web 與 Database 分手 1.7.2
- 前端靜態頁面暫存 1.7.3
- 頁面片段暫存 1.7.4
- 動態資料快取 1.7.5
- 增加 Web Server 1.7.6
- 增加 Database Server 1.7.7
- 分散式資料 1.7.8
- 增加更多的 Web Server 1.7.9
- Database 讀寫分離 1.7.10
- 高可用性的服務架構 1.7.11

常見問題 1.8
壓力測試 1.8.1
參考資料 1.9
書籍 1.9.1
文章 1.9.2
影片 1.9.3
工具 1.9.4

High Scaling Websites Structure Learning Notes 大型網站架構學習筆記

作者 : KeJyun

本書會介紹一些在建構大型網站中常用到的技巧或方法

書籍網址

項目	網址
Github Page	http://kejyun.github.io/high-scaling-websites-structure-learning-notes/
GitBook	http://kejyuntw.gitbooks.io/high-scaling-websites-structure-learning-notes/

聯絡資訊

項目	網址
Email	kejyun@gmail.com
LinkedIn	https://tw.linkedin.com/in/kejyun
Github	https://github.com/kejyun
Facebook	http://fb.me/kejyunTaiwan
Blog	http://blog.kejyun.com

所有 KeJyun 著作

- High Scaling Websites Structure Learning Notes 大型網站架構學習筆記
- Laravel 4 學習筆記
- Laravel 5 學習筆記
- SEO 學習筆記
- gulp 學習筆記
- Web Developer Learning Resource 網頁開發學習資源
- Mac OSX 新手入門
- Ruby on Rails 學習筆記

大型網站架構概念

當一個網站需要服務很多很多很多人的時候，我們就需要好的存取資料的架構，讓使用者能夠很快的拿到他想要拿的資料，這樣伺服器（Web & Database）才可以有更多「`時間`」及「`空間`」去服務更多的人。

你可能會聽到有一些在提高伺服器存取效率的關鍵字，像是「資料快取」、「資料庫索引」、「資料庫讀寫分離」、「資料分散式處理」...等等之類的方式。

提高效能存取的概念

不常異動的資料做快取

後端資料

以部落格文章為例，作者發表一篇文章後，除了文章內有一些小小的錯誤或錯字需要編輯修改，否則大部份的時間文章都是不會去異動的。

在動態去資料庫存取本篇文章時，我們可能會使用這樣的 SQL 語法去存取：

```
SELECT * FROM posts WHERE post_id = '部落格文章編號' Limit 1;
```

當每個使用者要來看這篇文章時，都需要透過這樣的 SQL 指令去撈取這篇文章的資訊（標題、內文...blabla），如果文章讀者不多時，偶爾這樣取資料庫撈資料還撐得過去，但像是比較熱門被轉載的文章，每分鐘可能有好幾百幾千人要去看這篇文章的資料，對於資料庫就是執行 1000 次這樣相同的語法去取得相同的資料（要記得，資料庫的存取是很昂貴的）。

為了減少資料庫存取次數，所以在程式方面我們會將這些不長異動的資料在第一次從資料庫撈取之後，把它 `快取` 下來，可能用 `Memcached`、`Redis` 或 `File` 的方式將資料預先存下來，然後再設定資料的過期時間，當超過過期時間後，再重複去資料庫撈取資料。

這樣可以減少資料庫的運算資源，使用者也可以很快地拿到他們想要看的資料，提高了伺服器對於資料的存取量。

前端資料

我們在網站上常常需要用 JavaScript 去完成我們要的 UI 操作效果，但是這些 JavaScript 除非操作方式有做大幅的異動，否則我們很少會修改這些 JavaScript 檔案（在 CSS 檔案部分也相同）。

除非有做特別的設定，再不重新整理畫面的前提下，瀏覽器都會幫我們的 JavaScript 及 CSS 檔案去做快取，載入過一次後就存放在使用者自己的電腦上，等到下次我們需要使用相同的 JavaScript 或 CSS 檔案時，瀏覽器會直接取用使用者電腦本地端的檔案，而不會去遠端伺服器再次拿取 JavaScript 或 CSS 檔案。

我們已可以利用瀏覽器的這個特性，等到 JavaScript 或 CSS 檔案做變更時，再透過 `版本號` 或 `不同的檔名` 去讓瀏覽器讀取最新的 JavaScript 或 CSS 檔案：

版本號

```
<script src="kejyun.app.js?v=1.0.0"></script>
<script src="kejyun.app.js?v=1.0.1"></script>

<link rel="stylesheet" href="kejyun.app.css?v=1.0.0">
<link rel="stylesheet" href="kejyun.app.css?v=1.0.1">
```

不同檔名

> 這個部分通常是用後端程式去控制 JavaScript 及 CSS 現在的版本檔名

```
<script src="kejyun.app-asdfghjkl.js"></script>
<script src="kejyun.app-qwertyuio.js"></script>

<link rel="stylesheet" href="kejyun.app-asdfghjkl.css">
<link rel="stylesheet" href="kejyun.app-qwertyuio.css">
```

索引

對於需要常用來查詢的資料做好索引，可以加快查詢的效率

資料庫讀寫分離

以部落格文章爲例，在 80/20 法則中，大部份 80% 人都是在看文章比較多（讀取資料：SELECT），只有少部分 20% 的人或意見領袖，才會發表文章表示看法（異動資料：INSERT、UPDATE、DELETE），而在做資料異動的時候很有可能會對資料進行鎖定，進而去影響讀取的速度。

> 除了有關交易（Transaction）的資料在 SELECT 的時候才有可能對資料進行鎖定，像是購票或購買限量商品時，會把撈取出來加入購物車的資料先行鎖定

所以像是 Facebook、部落格之類的媒體，大多會把資料庫做讀寫分離，使用者做異動的行爲會去主資料庫（Master）去做寫入的動作，然後從資料庫（Slave）在定期的去同步資料庫的內容，而其他大部份的讀者在讀取資料時，都去讀取從資料庫（Slave）的資料，這樣就不會有因爲資料異動而導致資料被鎖定，造成讀取變慢的問題。

而在主從架構中，可以是有很多台從的資料庫（Slave），透過分散式處理可以將不同的連線分配給不同的從資料庫（Slave），讓每一台從的資料庫平均分配差不多的連線量，因爲需要處理的連線減少了，進而讓每個查詢都能夠在短時間都能夠回應查詢結果，提高系統的可用度。

重要觀念

資料庫存取是很昂貴的

每下一段資料庫 SQL 語法去撈取資料，資料庫就要對資料去進行比對運算，當資料很龐大，SQL 語法又太複雜導致運算很繁瑣時，資料庫需要進行運算的時間又會變得更久，所以建立好的索引（Index）及下好的 SQL 語法是很重要的工作

> 雖然資料都撈得出來，但是撈得漂亮，撈得快也是一種藝術啊～

與資料庫建立的連線是很昂貴的資源

我們要對資料庫進行操作就需要先與資料庫進行連線，但是資料庫的連線建立是很花時間的，時間越久會導致回應的時間也會跟著變長，然而資料庫的連線數又不能無限制的擴張（需要看主機效能的乘載量），所以連線的資源就相對的珍貴。

我們通常會用連線池（Connection Pool）的方式去維持資料庫的連線，建立連線後做完查詢就將連線丟到連線池內，供下一個要做查詢的人使用，下一個要做查詢的人發現有可用的連線，就不用花費時間重新的去建立與資料庫的連線，直接使用就有的連線即可，直到這個連線時間超過資料庫最高的限制失效斷線爲止，這樣就能提高資料庫連線使用的效率。

對於需要當作查詢的資料必須建立索引

我們會依照我們想要撈取的資料下 `WHERE` 、 `GROUP BY` 、 `ORDER BY` ...等等的條件去撈取我們想要撈取的資料，如果不建立索引資料庫也是有辦法將資料撈出來，只是速度會比較慢，當這些條件資料有預先做索引時，資料庫可以很快地利用索引去完成查詢的動作

> 索引原理就是將資料透過資料結構（e.g. B-tree), Hash）預先做排序的存放，使用索引時資料庫可以很快的資料放在哪一個地方，以達到快速撈取資料的目的

以書籍爲例

索引就像書籍最前面的目錄一樣，可以很快地告訴我們哪一段章節的資料在第幾頁，所以我們就可以很快地翻閱到我們想看的資料，如果書籍沒有做目錄（索引）的話，我們要找到特定章節的資料時，我們就需要從頭到尾翻閱去查看看我們要看的資料到底在哪一頁，這對於

書籍的讀者來說是相當浪費時間的一件事。

詳細的索引說明會在稍後章節提到

結論

大部份提高存取效率的作法大多是：

設計好的程式邏輯

在做簡單的資料撈取，能夠用更快的方式（減少迴圈、快速比對資料...blabla）去產生我們要的資料，當然能夠加快請求的速度，但是在現在硬體處理的速度越來越快的情況下，程式的邏輯沒有太複雜下（複雜的可能像是演算法），大部份的資料處理都是很快的，所以大部份的瓶頸都是卡在資料庫資料的撈取邏輯、資料存放方式、資料撈取方式以及索引建置的邏輯。

減少資料庫的存取

資料能夠快取不要重複撈取就不要重複撈取，可以把資料存放在記憶體的快取中，若是整個頁面都很少異動，也可以把整個畫面（View）的 HTML 去做快取，把資料組合成 HTML 的動作都省下來了。

降低資料表資料量

當一個資料表只有 100 筆或是到 1000 筆資料的時候，資料庫大部份都的處理速度都是很快的（只有幾毫秒而已，人感覺不出來差異），但是當一個資料表的資料量有好幾百萬或好幾千萬筆資料，要從這麼多的資料去撈取出想要的資料，需要耗費的運算資源就會更多，所以讓資料能夠分散儲存，降低每個資料表的資料量，這樣也可以大大提高資料庫存取的效率。

以書籍為例

想像一下一本書有 100 頁或是到 1000 頁的量，我們在透過書籍目錄去查詢我們想要看的資料，應該速度不會差太多，但是當一本書有好幾百萬頁（現實上可能會分很多冊，像百科全書），我們需要看的書籍目錄也會比較多（索引表很大），所以也需要花更多時間去查詢我們要的資料。

資料庫分散式處理

當資料異動或讀取的資料量過大時，資料的異動會影響到讀取的效能，所以我們就需要將資料做讀寫分離。

建立良好的資料庫索引

將用來做資料撈取判斷的資料做好索引，而索引的順序及 WHERE 條件順序會影響索引的使用，這個部分會在之後提到。

參考資料

- B樹 - 維基百科，自由的百科全書
- 全文檢索 - 維基百科，自由的百科全書
- 80/20 法則

資料庫

在這裡會介紹一些如何提高查詢資料庫效率的相關技巧

資料排列組合

在使用 MySQL 或 PostgreSQL 資料庫的時候，我們常常會使用 Auto Increment 的整數去當作我們的鍵值，Int 總共有 `4294967295` 種排列組合方式，而 BigInt 則總共有 `18446744073709551615` 種排列組合方式

使用 Auto Increment 的好處是：

- 可以避免每次的鍵值都是不同的，不需要再做額外的欄位資料檢查是否重複
- 可以依照整數順序去判斷資料新增的順序

但這樣也有壞處：

- 資料被刪除後，空出來的鍵值很難被管理，會造成不必要的浪費
  - e.g. Queue 資料
- 總資料量被限制，當我們預期資料很有可能超過整數欄位的限制時，資料量超過時還需要處理溢位問題
  - e.g. 很多的 Queue 資料

所以若預期資料量很大時，我們會傾向使用字串去當作資料表的鍵值，單一整數的資料排列組合有 10 種 `（0-9）`，而單一字串的資料排列組合有 36 種 `（0-9 A-Z）`，雖然整數欄位在資料表搜尋的速度比字串快很多，但是這個是沒辦法的必要之惡。

字串排列組合

在資料庫儲存的字串資料是不區分大小寫的，所以若有一個 Primary 或 Unique 的鍵值，裡面是沒辦法同時儲存 `AA` 、 `Aa` 、 `aA` 或 `aa` 的，所以我們字串鍵值的排列組合只能用不區分大小寫 36 種 `(0-9 A-Z)` 排列組合去計算。

字串長度	排列組合 (不區分大小寫)
1	36
2	1,296
3	46,656
4	1,679,616
5	60,466,176
6	2,176,782,336

7	78,364,164,096
8	2,821,109,907,456
9	101,559,956,668,416
10	3,656,158,440,062,980
11	131,621,703,842,267,000
12	4,738,381,338,321,620,000
13	170,581,728,179,578,000,000
14	6,140,942,214,464,820,000,000
15	221,073,919,720,733,000,000,000
16	7,958,661,109,946,400,000,000,000
17	286,511,799,958,070,000,000,000,000
18	10,314,424,798,490,500,000,000,000,000
19	371,319,292,745,659,000,000,000,000,000
20	13,367,494,538,843,700,000,000,000,000,000
21	481,229,803,398,374,000,000,000,000,000,000
22	17,324,272,922,341,500,000,000,000,000,000,000
23	623,673,825,204,293,000,000,000,000,000,000,000
24	22,452,257,707,354,600,000,000,000,000,000,000,000
25	808,281,277,464,764,000,000,000,000,000,000,000,000
26	29,098,125,988,731,500,000,000,000,000,000,000,000,000
27	1,047,532,535,594,330,000,000,000,000,000,000,000,000,000
28	37,711,171,281,396,000,000,000,000,000,000,000,000,000,000
29	1,357,602,166,130,260,000,000,000,000,000,000,000,000,000,000
30	48,873,677,980,689,300,000,000,000,000,000,000,000,000,000,000
31	1,759,452,407,304,810,000,000,000,000,000,000,000,000,000,000,000
32	63,340,286,662,973,300,000,000,000,000,000,000,000,000,000,000,000
33	2,280,250,319,867,040,000,000,000,000,000,000,000,000,000,000,000,000
34	82,089,011,515,213,400,000,000,000,000,000,000,000,000,000,000,000,000
35	2,955,204,414,547,680,000,000,000,000,000,000,000,000,000,000,000,000,000
36	106,387,358,923,717,000,000,000,000,000,000,000,000,000,000,000,000,000,000
37	3,829,944,921,253,800,000,000,000,000,000,000,000,000,000,000,000,000,000,000
38	137,878,017,165,137,000,000,000,000,000,000,000,000,000,000,000,000,000,000,0

39	4,963,608,617,944,920,000,000,000,000,000,000,000,000,000,000,000,000,000,000,00(
40	178,689,910,246,017,000,000,000,000,000,000,000,000,000,000,000,000,000,000,(

> 只要字串長度大於 13 時，我們擁有的資料排列組合就會超過 BigInt 的 `18446744073709551615` 排列組合

在使用字串當做鍵值時，在我們每次新增資料手動產生字串鍵值時，我們都要再去資料庫檢查看看此鍵值是否存在，再新增進去，且我們需要額外做新增資料造成 Duplicate Key 的例外狀況處理

> 在我們字串很長時雖然發生碰撞的機率很低，但爲了系統資料的安全完整性，還是需要做檢查的步驟，並在頻繁被通知到鍵值衝突的狀況時，就要考慮增加字串長度，減少鍵值碰撞機率了

參考資料

- MySQL資料型態描述與比較
- Store UUID in an optimized way
- 《江南Style》如何衝破YouTube的計數器
- 資料庫索引數據結構及主鍵設計(b+tree)(part 1)

MySQL

在這裡會介紹一些在 MySQL 提高存取效率的一些技巧及方法

資料類型

越小越好

要使用能夠用來正確儲存資料的最小類型的資料型態，要確保不會低估要儲存的資料大小，因爲當資料可以確定用比較少的資料去儲存的時候，所 `使用的空間` 也會更少，所 `需要的索引空間` 也會越少，取得資料所需要的 `計算時間` 也會越短。

> 在可以用 `CHAR(20)` 去儲存資料時，就不要用 `CHAR(200)`
>
> 在可以用 `VARCHAR(20)` 去儲存資料時，就不要用 `VARCHAR(200)`
>
> 在可以用 `TINYINT` 去儲存資料時，就不要用 `INT`
>
> 在可以用 `TEXT` 去儲存資料時，就不要用 `LONGTEXT`
>
> 在可以用 `BLOB` 去儲存資料時，就不要用 `LONGBLOB`

越簡單越好

> 整數（INT）> 固定字串（CHAR）> 變動字串（VARCHAR）> 文字（TEXT）

越簡單的資料類型，資料庫所需要用來建立索引的效率越好，因爲資料類型越複雜代表資料的排列組合越多，所以需要更大的索引及計算去取得資料

> 在可以用整數（INT）去儲存資料時，就不要用固定字串（CHAR）
>
> 在可以用固定字串（CHAR）去儲存資料時，就不要用變動字串（VARCHAR）
>
> 在可以用變動字串（VARCHAR）去儲存資料時，就不要用文字（TEXT）

盡量不使用NULL

若非必要儲存NULL的資料，否則要盡可能的把資料欄位設定爲NOT NULL，資料庫很難最佳化有NULL資料欄位的查詢，可以NULL的資料欄位需要更多的儲存空間，資料庫還需要對其進行特殊處理，而當有NULL資料欄位使用 `索引（INDEX）` 的時候，每一條的索引紀錄必需要額外紀錄資料，導致查詢時索引的效率降低。

若眞的要儲存NULL，在 `不影響原有的資料` 的情況下，可以考慮用 `0` 、 `特殊值` ...等等之類的值去代替，可以用來區別是否爲NULL

結論

一切的資料類型都只能看自己應用的需求去決定，如果沒辦法還是要用比較複雜的資料類型，那還是必須要用，資料的完整性比任何的東西都重要多了，效率就想辦法用`增加機器`或者是`優化資料表結構`...等等的其他方式去達成，不要爲了效率增加而強迫使用特定的資料類型，這樣反而是因噎廢食。

參考資料

- KeJyun學習日誌: 提高存取MySQL效率小技巧

程式語法

指定使用索引

```
SELECT id
FROM data USE INDEX(type)
WHERE type=12345 AND level > 3
ORDER BY id
```

在下SQL語法的時候，有時某些語法使用某些索引執行效率會比較好，可是有時候MySQL沒辦法替我們選擇一個最適合的索引，導致執行的效率很慢（slow query），所以我們必須使用USE INDEX去指定執行效率好的索引，以提升效率。

少用 JOIN，多用幾次 SELECT 撈取大量資料

```
SELECT user.id , user.name , post.id , post.content
FROM user , post
WHERE user.id = post.user_id ......
```

我們在撈取使用者的文章資訊的時候，我們可能會用 JOIN 去撈取我們要的資料，這樣很直覺，只是當使用者資料有 `10000` 筆，而文章有 `99999` 筆，像這種有大量資料時候的話，使用 JOIN 對於資料存取真的是惡夢，因爲 JOIN 過後表示會有 `10000x99999=999990000` 筆資料，然後再從這麼大量的資料中去撈取 WHERE 判斷式中指定的資料，在資料庫伺服器記憶體不夠的時候鐵定會炸掉。

解決方式是我們可以分批撈取使用者的資料，以及文章的資料

```
SELECT user.id , user.name
FROM user
```

```
SELECT id , content , user_id
FROM post
```

在使用者資料資料撈取出來之後，使用迴圈將使用者存成陣列，但是「陣列的索引」是使用可以識別的「使用者編號（user.id）」當作索引值。

```
<?php
$user['user.id'] = array();
```

在撈取文章資料後，必須將撈取的文章指定給該文章的作者（使用者），而我們撈取的使用有撈取「使用者編號（user.id）」，所以可以透過這個資料，將文章存到使用者資料下的文章陣列。

```
<?php
$user[$post['user_id']]['post'] = $post_array;
```

透過分次撈取，本來的 `10000x99999=999990000` 筆資料，就會變成 `10000+99999=109999` 筆資料，記憶體消耗降低極大下，也可以達到同樣的效果

後記

JOIN 在資料庫幾乎是必學的語法，讓資料表的正規化，避免資料冗余，在查詢的時候也可以透過 JOIN 組合出想要的資料，所以 JOIN 對於資料庫正規化有它的用途，但資料庫的發展的初期中，還沒有像現在有用到巨量的資料（Google、Facebook、Twitter ... etc），所以在流量&資料量小的時候，資料庫在 JOIN 時的查詢回應速度通常差沒幾毫秒 （0.01~0.05毫秒），所以都在可接受的範圍，但現在大型網站一天都好幾千萬的存取量，假如每天有 1000 萬的存取次數，而每個地方都慢了 0.05 毫秒，總共就會慢了 `10,000,000 x 0.05 = 500,000 毫秒 = 500 秒` ，在我們要求每次回應時間最差都要在 1 ~ 2 秒內回應，若是像 Google、Facebook 這樣的大型服務，要求可能都要在 0.1 秒以內，這樣慢的時間可能會讓使用者等到不耐煩，若 $$ 多多加機器僅能簡單應付巨量級的查詢，沒辦法解決 JOIN 所造成的查詢瓶頸，所以在巨量級資料的情況下，幾乎很少會使用 JOIN 的方式去查詢所有的資料出來，天下武功，無堅不摧，唯快不破。

更新或刪除大量範圍的資料，請用主鍵去更新

我們可能會想要一次異動大範圍的資料，若以社群網站的 `通知（Notification）` 來說，我們希望在使用者點選通知列表後，將所有的通知從未讀狀態變更爲已讀，我們可以很容易地想到用這樣的 SQL 語法去進行更新：

```
UPDATE `notification` SET status="已讀" WHERE created_at < '現在時間';
```

若我們要刪除大量的資料時，我們也可能用這樣的 SQL 語法去刪除資料：

```
DELETE FROM `backup` WHERE created_at < '現在時間';
```

但是在這樣的條件爲範圍的語法，資料庫會需要一筆一筆的去比對資料是否爲設定條件的範圍，若爲 `交易` 的資料，資料庫會將資料進行鎖定禁止讀取，有可能鎖定一些非我們要處理的資料，若處理時間過久，可能會導致撈取資料的反應時間過久。

所以通常爲了避免 `鎖定不必要資料` 的情況發生，我們可以試著先把資料撈取出來，再透過 `明確的主鍵` 去指定哪些異動的資料需要被鎖定，而不影響其他的資料。

更新資料（較佳作法）

```
-- 先撈取資料
SELECT `notification` WHERE status="未讀" AND created_at < '現在時間';

-- 指定要更新資料的主鍵
UPDATE `notification` SET status="已讀" WHERE  id IN (1, 2, 3);
```

刪除資料（較佳作法）

```
-- 先撈取資料
SELECT `backup` WHERE created_at < '現在時間';

-- 指定要刪除資料的主鍵
DELETE FROM `backup` WHERE id IN (1, 2, 3);
```

在服務會有大量的人撈取資料的時候，這樣可以避免不必要的資料鎖定，也可以讓存取速度加快喔～

參考資料

- KeJyun學習日誌: 提高存取MySQL效率小技巧

索引

索引順序

在所有的資料庫內，建立索引是提昇資料庫資料存取效率的很重要的方式，但是錯誤的 `索引順序` 或是 `SQL 語法` 都可能讓查詢語法變成了 slow query。

假設我們有一個部落格文章的資料表，裡頭存放所有使用者的部落格文章，資料表資料如下：

posts

post_id	post_status	author	content
1	正常	kejyun	文章 1
2	刪除	kejyun	文章 2
3	正常	kejyun	文章 3
4	正常	jimmy	文章 4
5	刪除	jimmy	文章 5

如果我們要撈取 `作者為 kejyun` 且 `狀態正常` 的文章，我們可能會用下列語法去撈取

SQL 語法 1

```
SELECT *
FROM posts
WHERE post_status = '正常'
AND author = 'kejyun'
```

或者用這個語法撈取

SQL 語法 2

```
SELECT *
FROM posts
WHERE author = 'kejyun'
AND post_status = '正常'
```

這兩句 SQL 語法都可以撈出我們想要的結果，但是對於不同的索引執行的效率卻是差異很多

如果我們的索引是 `post_status` 、 `author` 的順序， `SQL 語法 1` 則會正確的使用索引來做查詢，執行的效率會很快。

但是對於 `SQL 語法 2` ，會因爲找不到適合該語法的索引去做查詢，所以會變成不使用索引，而對整個資料表作完整的檢索，去查找出資料來。

主要原因是

> 在 SQL 語法 WHERE 的順序會影響查詢索引的順序
>
> 同樣的索引資料在不同的順序，表示爲不同的索引，所以 `post_status、author` 與 `author、post_status` 這兩個索引雖然是使用相同的資料欄位，但因爲順序不同所帶來的查詢效果也不同
>
> 同一句 SQL 語法只能使用單一個索引，所以不同的索引沒辦法共用

資料庫沒有這麼人工智慧，會去判斷兩個 SQL 語法可以使用相同的索引去查找出相同的資料，他會依照程式設計師給的語法，依序去查找是否有可用的索引。

分析 **SQL** 語法 **1**

在 `SQL 語法 1` 中，我們 WHERE 的第 1 個條件是 `post_status = '正常'` ，所以資料庫會去查找有沒有索引開頭是使用 `post_status` 的索引。

若有該索引，則再判斷 WHERE 的第 2 個條件 `author = 'kejyun'` ，所以資料庫會去查找有沒有索引開頭是使用 `post_status` 的索引，且該索引的第 2 個索引是使用 `author` 的索引。

資料庫會一直判斷比較 SQL 語法 WHERE 條件與索引之間的順序關係，直到沒辦法匹配後，後面的沒辦法匹配索引則使用完整比對的方式去進行查詢

分析 **SQL** 語法 **2**

在 `SQL 語法 2` 中，我們 WHERE 的第 1 個條件是 `author = 'kejyun'` 但我們資料庫沒有建立索引開頭是使用 `author` 的索引，所以只能對資料庫使用完整比對。

原理

索引就像是書籍的目錄一樣，若以童話故事書爲例，我們書中會收錄世界各國的故事，且每個故事有他自己的類型，像是奇幻、驚悚、傳説、神話。

如果我們以國家當做大標題（e.g. 第 1 索引），以故事類型當作小標題（e.g. 第 2 索引），那麼書籍目錄會像：

- 台灣
  - 奇幻
  - 驚悚
  - 傳說
  - 神話
- 日本
  - 奇幻
  - 驚悚
  - 傳說
  - 神話
- 歐美
  - 奇幻
  - 驚悚
  - 傳說
  - 神話
- 印度
  - 奇幻
  - 驚悚
  - 傳說
  - 神話
- etc...

如果我們要找 `台灣` 且類型爲 `傳說` 的童話故事，我們的目光會先移動到台灣的區段，然後在這個區塊下找到類型爲 `傳說` 的故事

- 台灣
  - 奇幻
  - 驚悚
  - 傳說
  - 神話

但是如果我們要找所有類型爲 `傳說` 的故事中，發生在 `台灣` 的故事，依照步驟我們會希望先把所有 `傳說` 的故事先列出來，再從這個目錄下去找屬於 `台灣` 的故事，所以我們希望會看到像這樣的目錄（索引）：

- 傳說
  - 台灣
  - 日本
  - 歐美
  - 印度

但是在這本故事書的目錄（索引）中，我們沒有看到這樣的目錄結構，所以我們沒辦法透過目錄快速的找到我們要看的所有類型爲 `傳說` 的章節資料，只好從頭到尾的去翻閱整本書，直到找到全部我們 `傳說` 章節的故事，然後再從這些找出來的 `傳說` 故事中，再去區別出哪些爲 `台灣` 的故事。

所以目錄（索引）的規則就是希望看故事書的人要怎麼快速找到他想要的東西，當沒有我們可以參考的目錄的話，就像資料庫沒有可參考的索引一樣，就會找得比較慢（但還是找得出來）。

結論

因爲索引順序的不同，以及 SQL WHERE 條件順序的不同，會使得資料庫在使用索引進行查詢有不同的效率，所以要謹慎的使用索引及 SQL 語法，才能達到高效率的查詢結果。

控制索引更新

關閉索引更新：

```
ALTER TABLE table_name DISABLE KEYS;
```

開啓索引更新：

```
ALTER TABLE table_name ENABLE KEYS;
```

MySQL在新增（INSERT）、刪除（DELETE）、更新（UPDATE）的時候會去更新現有的索引表，而更新索引表也需要花費一些時間，當異動一筆資料的時候，索引表也做一次的異動，但當在做大量資料異動的時候，例如異動1000筆資料，索引表也需要異動1000次，而其實我們只需要最後一次（最新）的異動就好了，前面的999次都是不需要做的索引表異動更新，所以在異動大量資料前，可以使用指令 `ALTER TABLE table_name DISABLE KEYS;` 關閉索引更新，等異動完成後，再使用指令 `ALTER TABLE table_name ENABLE KEYS;` 開啓索引更新。

```
ALTER TABLE users DISABLE KEYS;
異動（INSERT、DELETE、UPDATE）大量資料SQL語法
ALTER TABLE users ENABLE KEYS;
```

自定義Hash Index做字串完整比對

我們知道在對 `字串（CHAR或VARCHAR）` 去做查找的時候效率會遠比對 `整數（INT）` 查找還慢，因CRC32對字串做校驗後會回傳 `整數的校驗碼` ，我們在資料表增加一個整數型態欄位，儲存要比對字串的校驗碼。

crc32

> 建立str 的 32 位迴圈冗余校驗碼多項式。這通常用於檢查傳輸的資料是否完整。
>
> 由於 PHP 的整數是帶符號的，許多 crc32 校驗碼將返回負整數，因此你需要使用 sprintf() 或 printf() 的「%u」格式符來取得表示無符號 crc32 校驗碼的字串。 在原本實作email登入時會對email欄位做索引，所以會先去查找email字串欄位的資料，之後再去比對密碼是否正確，但若資料過多字串比對的效率會降低很多

```
SELECT id,name,email
FROM user
WHERE email = "kejyun@gmail.com"
AND password = "xxx"
```

我們加入了 `emailcrc` 的欄位去儲存對 email 字串的校驗碼，再查找 email 字串欄位的資料前，先透過crc整數校驗碼快速過濾掉不可能的資料，之後再從少數的資料中做 email 字串欄位字串比對，如果資料量很大，這樣的效率會提升很多。

```
SELECT id,name,email
FROM user
WHERE emailcrc = CRC32("kejyun@gmail.com")
AND email="kejyun@gmail.com"
AND password="xxx"
```

這邊要注意的是沒辦法只使用 crc 校驗碼去當作唯一的條件， `不同的字串可能會出現相同的校驗碼` ，所以最後還是要對你要比對的字串做比對，避免查詢發生錯誤。

email	emailcrc
kejyun1@gmail.com	1234567890
kejyun2@gmail.com	1234567890

參考資料

- KeJyun學習日誌: 提高存取MySQL效率小技巧
- KeJyun學習日誌: MySQL效率調校
- MySQL Indexing: Best Practices Slide PDF
- Tools and Techniques for Index Design PDF Slide

- EXPLAIN Demystified PDF slide
- Optimizing MySQL Configuration PDF Slide
- PHP手冊 - crc32

鎖表 (Lock Table)

定義

在要更新資料表的資料時，`MySQL` 會將表資料鎖定無法讀取，直到資料異動完畢，`MyISAM` 預設支援 `Table-level lock` ，而 `InnoDB` 預設支援 `Row-level lock`

Table-level lock

資料表資料異動時，將「`整個資料表（Table）`」都鎖定住無法讀取

Row-level lock

資料表資料異動時，將「`要更新的資料列（row）`」都鎖定住無法讀取

注意事項

在使用 `Row-level lock` 時必需要 `明確指定要異動資料的主鍵（Primary Key）` ，否則將會改用 `Table-level lock` 去做資料表的異動

範例

假設有 user 資料表，裡面有 id 與 name 的欄位，id 是主鍵

SQL	Table lock	Row lock	No lock	備註
SELECT * FROM user WHERE id='1' FOR UPDATE;	-	v	-	明確指定主鍵，並且有此筆資料，row lock
SELECT * FROM user WHERE id='-1' FOR UPDATE;	-	-	v	明確指定主鍵，若查無此筆資料，無 lock
SELECT * FROM user WHERE name='KeJyun' FOR UPDATE;	v	-	-	無主鍵，table lock
SELECT * FROM user WHERE id<>'1' FOR UPDATE;	v	-	-	主鍵不明確，table lock
SELECT * FROM user WHERE id LIKE '3' FOR UPDATE;	v	-	-	主鍵不明確，table lock

備註

FOR UPDATE 僅適用於 InnoDB，且必須在交易區塊(BEGIN/COMMIT)中才能生效。

參考資料

- KeJyun學習日誌: MySQL鎖表(Lock Table)Table-level與Row-level比較

交易

控制InnoDB Transaction

關閉自動提交：

```
SET autocommit=0;
```

開啓自動提交：

```
SET autocommit=1;
```

在 MySQL InnoDB 預設所有的資料異動都是 `Transaction（交易）`，當與資料庫做連線的時候，InnoDB 會採用 `自動提交（autocommit）` 的方式，所以除非使用 `BEGIN;` 及 `COMMIT;` 將異動語法包覆起來設定爲同一個 ***Transaction***，否則任何一個資料異動的語法（***INSERT***、***DELETE***、***UPDATE***）會認爲是一個獨立的 ***Transaction***。

所以資料表每次做資料異動的時候會一直 `提交（COMMIT）` 去更新日誌，若有 1000筆 獨立的 SQL要執行就會被 COMMIT 1000 次，所以在下異動語法之前，可以使用指令 `SET autocommit=0;` 關閉自動提交，等異動完成後，再使用指令 `SET autocommit=1;` 開啓自動提交。

```
SET autocommit=0;
異動（INSERT、DELETE、UPDATE）大量資料SQL語法
SET autocommit=1;
```

參考資料

- KeJyun學習日誌: MySQL效率調校

伺服器

設定

max_allowed_packet：允許最大封包

在INSERT或UPDATE的時候，因爲封包資料過大，導致錯誤發生

max_connect_errors：最大連線錯誤

設 4294967295，可以避免當 client (像是 php) 發生太多錯誤時被 block 住。

> default 0，請設定一個最大的值即可，32位元的系統 最大是 4bytes 4294967295、64位元的系統最大是 18446744073709547520

參考資料

- note: mysql 跟效能有關的設定 @ mini box 迷你小盒子 :: nidBox親子盒子

Explain 做 SQL SELECT 語法效能測試

在MySQL我們在使用 SELECT 做撈取資料的時候，有時候常常會效能低落，撈取資料需要很長的時間，有時候是 SQL 語法下得不好導致沒有使用到正確的索引去撈資料，我們這個時候就必須要檢查我們下的 SQL 語法到底有哪些地方需要改善，我建立的 comment 的資料表並新增幾筆假資料去做示範

```
-- 建立資料表

-- 留言
CREATE TABLE IF NOT EXISTS `comment` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT COMMENT '編號',
  `content` varchar(50) COLLATE utf8_unicode_ci NOT NULL COMMENT '留言',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci AUTO_INCREMENT=1 ;

-- 使用者
CREATE TABLE IF NOT EXISTS `user` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT COMMENT '編號',
  `name` varchar(30) COLLATE utf8_unicode_ci NOT NULL COMMENT '姓名',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci AUTO_INCREMENT=3 ;

-- 使用者的留言
CREATE TABLE IF NOT EXISTS `user_comment` (
  `user_id` int(10) unsigned NOT NULL COMMENT '使用者編號',
  `comment_id` int(10) unsigned NOT NULL COMMENT '評論編號',
  PRIMARY KEY (`user_id`,`comment_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;



-- 新增資料

-- 留言
INSERT INTO `comment` (`id`, `content`) VALUES
(1, '留言1'),
(2, '留言2');

-- 使用者
INSERT INTO `user` (`id`, `name`) VALUES
(1, '使用者1'),
(2, '使用者2');

-- 使用者的留言
INSERT INTO `user_comment` (`user_id`, `comment_id`) VALUES
(1, 1),
(1, 2);
```

```
-- 解釋MySQL語法效能

-- 撈取留言資料
EXPLAIN SELECT * FROM  `comment` WHERE  id` =2;

-- 撈取使用者的留言資料
EXPLAIN SELECT *
FROM  `comment` c,  `user` u,  `user_comment` uc
WHERE u.`id` = uc.`user_id`
AND uc.`comment_id` = c.`id`
```

解釋 MySQL 語法效能：撈取留言資料

+ 選項

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	comment	const	PRIMARY	PRIMARY	4	const	1	

解釋MySQL語法效能：撈取留言資料

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	c	ALL	PRIMARY	NULL	NULL	NULL	2	
1	SIMPLE	uc	index	PRIMARY	PRIMARY	8	NULL	2	Using where; Using index; Using join buffer
1	SIMPLE	u	ALL	PRIMARY	NULL	NULL	NULL	2	Using where; Using join buffer

而EXPLAIN後的資料有下面這些欄位

select_type

table

關連到的資料表

type

使用關聯查詢的類型(效率由好至壞排序)

- System
- const
- eq_ref
- ref

- fulltext
- ref_or_null
- index_merge
- unique_subquery
- index_subquery
- range
- index
- ALL

possible_keys

可能使用到的索引，從WHERE語法選擇出一個適合的欄位

key

實際使用到的索引，如果爲NULL，則是沒有使用索引

key_len

使用索引的長度，長度越短 準確性越高

ref

顯示那一列的索引被使用，一般是一個常數(const)

rows

MySQL用來返回資料的筆數，可以簡單的把rows視爲執行效能，越少越好

Extra

MySQL用來解析額外的查詢訊息

- Distinct

當MySQL找到相關連的資料時，就不再搜尋。

- Not exists

MySQL優化 LEFT JOIN，一旦找到符合的LEFT JOIN資料後，就不再搜尋。

- Range checked for each Record(index map:#)

無法找到理想的索引。此爲最慢的使用索引。

- Using filesort

當出現這個值時，表示此SELECT語法需要優化。因爲MySQL必須進行額外的步驟來進行查詢。

- Using index

返回的資料是從索引中資料，而不是從實際的資料中返回，當返回的資料都出現在索引中的資料時就會發生此情況。

- Using temporary

同Using filesort，表示此SELECT語法需要進行優化。此爲MySQL必須建立一個暫時的資料表(Table)來儲存結果，此情況會發生在針對不同的資料進行ORDER BY，而不是GROUP BY。

- Using where

使用WHERE語法中的欄位來返回結果。

- System

system資料表，此爲const連接類型的特殊情況。

- Const

資料表中的一個記錄的最大值能夠符合這個查詢。因爲只有一行，這個值就是常數，因爲MySQL會先讀這個值然後把它當做常數。

- eq_ref

MySQL在連接查詢時，會從最前面的資料表，對每一個記錄的聯合，從資料表中讀取一個記錄，在查詢時會使用索引爲主鍵或唯一鍵的全部。

- ref

只有在查詢使用了非唯一鍵或主鍵時才會發生。

- range

使用索引返回一個範圍的結果。例如：使用大於>或小於<查詢時發生。

- index

此爲針對索引中的資料進行查詢。

- ALL

針對每一筆記錄進行完全掃描，此爲最壞的情況，應該盡量避免。

結論

MySQL 的 Explain 可以分析大部份的 SQL 語法效能，但有些語法像是 WHERE IN 則會被歸類爲 range 的語法，但實際上則是 Using Where，所以確切的語法分析要再看看文件眞正的用法去決定

參考資料

- KeJyun學習日誌: 在MySQL使用Explain做SQL SELECT語法效能測試

程式

在這裡會介紹程式中要怎麼控制去快速的處理資料，提高程式的存取效率

陣列

鍵值（key）

在撈取資料庫的資料後，我們會把資料存在陣列中，等之後比對資料時可以當作查詢之用，若可以的話建議陣列的 `鍵值（key）` 要用可以辨識資料的值。

像是使用者的資料可以用使用者編號當鍵值，若之後要直接取特定使用者的資料，直接就可以指定使用者編號去取得使用者的資料：

```
<?php

$users[$user_id] = $user_info;
```

像是我們要取所有使用者的留言，留言資料中有使用者編號，不同的留言可能來自同一個使用者，但我們都要取用他的使用者名稱，這樣我們就可以很方便的取得該使用者的資料。

```
<?php

$messages = [
  [
    'id'      => 1,
    'user_id' => 1,
    'message' => 'Hi'
  ],
  [
    'id'      => 2,
    'user_id' => 2,
    'message' => 'hello'
  ],
  [
    'id'      => 3,
    'user_id' => 1,
    'message' => 'How are you?'
  ]
];

// 取得留言使用者姓名
foreach ($messages as &$message) {
  $message['user_name'] = $users[$message['user_id']]['user_name'];
}
```

快取

在這裡會介紹怎麼樣用快取去加快資料存取的效率

設定快取

對於同樣不常異動的資料，若有大量使用者存取，我們通常爲把資料設爲快取（可以存放在記憶體或檔案中），在下次有其他使用者要來存取相同資料的時候，不去資料庫重新撈取資料，直接將快取的資料回傳給使用者，可以減少資料庫的存取，增加系統處理使用者請求的效能。

以部落格文章快取爲例

我們會透過文章編號存取部落格文章，像是 `/posts/1` 或 `/posts/2`，以 PHP Laravel Framework 爲例，我們可能透過這樣的方式去處理存取部落格文章：

```
<?php

// 快取 key 依照不同的文章編號，可以是 blog_post:1 或 blog_post:2
$cache_key = 'blog_post:' . $post_id;

// 快取時間爲 60 分鐘
//（在 Laravel 的快取方法時間是以分鐘爲計，若是原生的快取方法則是以秒數爲計，所以要存放 60 分鐘則必須設定爲 60 * 60 = 3600）
$cache_minutes = 60;

if (Cache::has($cache_key)) {
  return Cache::get($cache_key);
}

$results = DB::select('SELECT * FROM `posts` WHERE `post_id` = ?', [$post_id]);

if ($result !== null) {
  // 若有找到該文章資料，則設定快取，讓下一位使用者再次存取相同文章時可直接取用快取
  Cache::put($cache_key, $results, $cache_minutes);
}

return $result;
```

我們會先判斷部落格文章的快取是否存在，若有則直接回傳快取資料，若沒有則重新去資料庫撈取文章資料，在撈取完資料後則設定該文章資料的快取，這樣的話之後有其他使用者要存取相同的部落格文章，則直接從快取拿資料就好，不用再到資料庫去撈取資料，減少資料庫的存取，以提升快取的存取效率。

快取的失效時間必須要依你自己的應用去設定，因爲快取大多是放在的記憶體當中（Memcached、Redis）這樣 I/O 存取速度才會快，但是記憶體是 `有限` 的昂貴資源，除非資料需要一直做快取讓使用者存取，否則快取通常不會設定過長的失效時間，希望失效後自動

清除並釋放記憶體資源給其他的快取或應用做使用。

若 $ 夠多，能夠買夠多的記憶體，將快取失效時間設爲 1 個月，甚至是更大也沒差，端看自己的 `硬體狀況` 及 `資料需要存放狀況` 去決定快取的時間。

刪除快取

我們設定了快取希望使用者存取資料的時候能夠加快回應時間，但資料可能會因爲使用者的 `刪除` 或 `修改` 而產生了異動，若我們需要讓使用者能夠讀取到最新的資料，我們通常會手動的將快取刪除，讓使用者下次讀取的話能夠去資料庫讀取到最新的資料，在重新設定一次新的快取資料。

以部落格文章快取爲例

更新部落格文章資料

```
<?php

// 快取 key 依照不同的文章編號，可以是 blog_post:1 或 blog_post:2
$cache_key = 'blog_post:' . $post_id;

$results = DB::update('UPDATE `posts` SET `content`= ? WHERE `post_id` = ?', [$new_con
tent, $post_id]);

if ($result) {
  // 選擇1: 直接設定文章快取，讓下一位使用者再次存取相同文章時可直接取用快取
  Cache::put($cache_key, $new_content, $cache_minutes);

  // 選擇2: 刪除快取，等下一位使用者讀取文章時在去設定快取
  Cache::forget($cache_key);
}

return $result;
```

刪除部落格文章資料

```php
<?php

// 快取 key 依照不同的文章編號，可以是 blog_post:1 或 blog_post:2
$cache_key = 'blog_post:' . $post_id;

$results = DB::delete('DELETE FROM `posts` WHERE `post_id` = ?', [$post_id]);

if ($result) {
  // 刪除快取，讓下一位使用者不要再讀取已被刪除的文章
  Cache::forget($cache_key);
}

return $result;
```

檔案

我們可能會需要產生圖片或檔案供使用者去下載，這些讓使用者可以公開存取檔案的原則大概有這些：

- 隨機檔名
- 同一目錄下不要有太多檔案
- 資料庫僅儲存圖片檔名，不要紀錄完整路徑
- 使用 CDN 提高負載平衡及降低頻寬成本

隨機檔名

很多人可能會圖方便，讓檔案用可以猜出規則的方式做儲存，例如有人可能在儲存會員的照片，是用會員的編號當作圖片檔名，如果會員編號是整數的欄位，則就更容易讓人家猜出來，更方便地去解析你主機的所有會員照片，像是：

```
100000001.png
100000002.png
100000003.png
...
100999999.png
```

所以通常我們會將圖片的名稱依照會員編號的規則去做加密，我們可能可以用 md5 去做加密的動作，這樣圖片檔名可能會像是：

```
d69d3401d01a8ceed4549434e5ad9f40.png (`100000001` md5)
5a5ea04157ce4d020f65c3dd950f4fa3.png (`100000002` md5)
154666111ab87ce13405f384faeb5428.png (`100000003` md5)
...
e0b50eb5c4add211417977e1f79364e0.png (`100999999` md5)
```

這樣圖片的檔名看起來隨機了許多，比較不容易被猜出來，但是對比較厲害的人，可能會想要拿你的會員編號自己做 md5，所以還是可以看出規則，這時候我會試著在 md5 的會員編號字串加自定義的 salt 字串像是 `KeJyun` ，這樣出來的圖片檔名會像這樣：

```
a3bce35e2b4e57911869b1bf5d040d71.png (`100000001KeJyun` md5)
5121a0fabb52c26c01817ee65df683a8.png (`100000002KeJyun` md5)
2ec12af8e6e011a0f5256b61db2145c3.png (`100000003KeJyun` md5)
...
467bd65bf6d4864adcc633caf2a70f7d.png (`100999999KeJyun` md5)
```

這樣除非對方知道我們使用哪一個 salt 去做加密，否則就很難被猜出來

同一目錄下不要有太多檔案

我們存放會員大頭照的路徑可能會像是：
`http://www.kejyun.com/users/images/a3bce35e2b4e57911869b1bf5d040d71.png`

在上面的例子我們把所有的會員照片都存放在 `users/images` 的目錄下，對於系統程式，我們可以很快的指定路徑去讀取該檔案出來，但如果對於系統端的管理者，要去管理這些檔案時就會造成很大的困擾，想像你有 1000 萬個會員照片，所有的會員照片都放在同一個目錄下面，光是要使用 `ls` 指令去列出那個目錄的所有照片，對於系統來說就是一大夢魘...

所以我們通常會對檔案做資料夾的分層處理，像是 `a3bce35e2b4e57911869b1bf5d040d71.png` 檔案，我們可能取其前 3 個字元 `a3b` 去做資料夾的分層，所以我們圖片會份在 `users/images/a3b/a3bce35e2b4e57911869b1bf5d040d71.png` 目錄下，分層資料夾的 3 個字元的資料排列組合爲 `46,656` 種，所以若我們有 `1000 萬` 個會員照片，我們可以預期照片會被平均的放在到各個分層資料夾，每個分層資料夾約 `214 張` 照片 `（10000000 / 46625 = 214.33）`，這樣對於系統在做檔案的索引及瀏覽也會比較快。

資料夾的分層要如何分層，完全取決於你系統專案資料數量的需求，若你預期你的檔案會超過 1 億種，那麼分層方式我們可能會分成 2 層，依照上面規則，照片可能會被放到 `users/images/a3bc/e35/a3bce35e2b4e57911869b1bf5d040d71.png` 下面，要怎麼分層處理完全取決於你的需求狀況，沒有一定的規則！

資料庫僅儲存圖片檔名，不要紀錄完整路徑

我們的檔案可能會隨著系統使用人數變多，需要轉換成不同的管理方式，所以檔案在做大型專案時，很有可能會隨時被移動位置的，所以千萬不要將完整的圖片路徑存到資料表欄位中，否則以後在做檔案效能最佳化調校時，資料表檔案路徑還需要重新的異動更新，會是很大的難題，可能的瓶頸可能會卡在：

- 專案不能只爲了調整圖片路徑而關站維護
- 資料量很大，更新檔案路徑很慢，可能需要 3~7 天以上去更新資料表的資料
- 若更新失敗，或者是又需要重調整架構時，所耗費時間會更長

所以我們在資料表只要紀錄檔案的檔名就好，所有與架構有關的資料通通由程式去控制產生，程式只要改一下，檔案的存取架構隨時都可以容易的做異動

使用 CDN 提高負載平衡及降低頻寬成本

我們的主機可能會使用 AWS、Linode 之類的雲端服務，這些的主機可能是放在日本或美國，我們希望使用者去距離使用者自己比較近的 CDN 主機，去存取這些不常被異動的靜態資料，這樣可以提高原本主機本身的負載流量，將流量做分散式的處理。

我們的圖片可能放在像是這樣的網址下:

```
http://cdn1.kejyun.com/users/images/a3bc/e35/a3bce35e2b4e57911869b1bf5d040d71.png
http://cdn2.kejyun.com/users/images/a3bc/e35/a3bce35e2b4e57911869b1bf5d040d71.png
http://cdn3.kejyun.com/users/images/a3bc/e35/a3bce35e2b4e57911869b1bf5d040d71.png
...
http://cdn9.kejyun.com/users/images/a3bc/e35/a3bce35e2b4e57911869b1bf5d040d71.png
```

這些不同的主機存放的圖片都與原主機的 `http://www.kejyun.com/users/images/a3bc/e35/a3bce35e2b4e57911869b1bf5d040d71.png` 相同，只是我們把同樣的靜態檔案分流複製到各個 CDN 主機上，這樣就不用讓原主機承載過高的頻寬吞吐量了！

參考網站

- 內容傳遞網路 CDN - Wiki
- CloudFlare

網站架構演進

這裏會介紹從小型網站到大型網站整個的架構，是怎麼規劃及演進的。

Web 與 Database 在一起

把 Web 或 Database 弄在同一部機器這樣的方式，通常是用在程式開發的「本地測試機」中，或者通常不會 `同時` 拿來服務太多的人的應用（共同上線人數），像是一般基本的公司形象網站、個人介紹頁...等等不會有太多人同時存取的資料懷鏡時才會使用這樣的架構。

除了基本的 Apache（Nginx）及 MySQL（Postgres），視情況需要設置會把其他的服務也放在同一台機器，像是 Memcached、Redis、Node.js。

架構圖：

Web Server / Database

使用時機

同時上線人數約 10 ~ 15 人左右

人數爲 ***KeJyun*** 過去經驗大概估算的人數，沒有經過實際測試僅參考用

當一台機器需要乘載多種服務時，同樣的主機資源要做很多事情，當然處理的效率通常都比較低

以飲料店來當例子，1 個服務生可以同時服務 3 位客人（主機同時有 3 個人發出請求），同時替客人做點餐結帳、做飲料的工作，但是當客人越來越多時來到 15 人的時候，因爲人的能力有限（主機的效能有限），所以後面很多人就需要排隊等前面幾位客人拿到飲料後才能繼續服務，所以就需要等待排隊（主機回應時間拉長），所以要看自己的經濟規模（同時乘載規模）去聘請適當人數的員工（規劃使用不同的主機架構）

> 同時上線非每日上限，同日上線指的是同一時先有多個人同時跟伺服器要資源

服務的可用度不需要太高

大部份的使用者可能最高能夠忍受等待一眨眼的時間（約 250 毫秒 = 0.25 秒），但是這個標準是在產品一秒鐘幾十萬上下時才需要達到的可用度標準（像是股票交易、電子商務網站、社群網站...等等），一般只是測試用或是形象的網站不需要那麼高的可用度，能夠在 1 ~ 2 秒內回應都是在可接受的範圍（當然不能太久超過 4、5 秒以上，使用者會以爲網站掛掉了沒有回應）

資料有做異地備援

資料可以說是整個企業的資產及生命，當資料自己有做異地備援的話，就可以用這樣的方式，不然主機太操很容易掛掉（就像人一樣加班太久沒休息，就很容易生病），裡面的資料可能會因爲沒辦法救回，所以若是有做資料備份，不怕資料不見的狀況下再去用這樣的架構去做服務。

參考資料

- 你願意花多久等待網站回應？ 一眨眼都嫌太久了
- 瞭解網站速度 - Analytics (分析) 說明

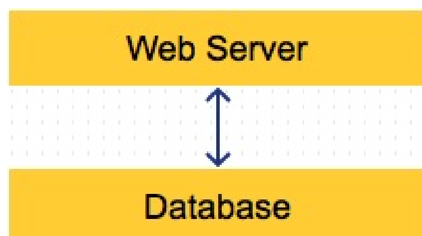
Web 與 Database 分手

當服務越來越多人需要去同時存取時，原本的 `Web` 與 `Database 在一起` 的架構會變得不敷使用了，網站的 `反應時間` 會變得越來越慢，伺服器的 `CPU` 及 `記憶體` 的消耗也會變得越來越高。

而且服務彼此還會互相搶機器的資源，若機器發生問題導致資料庫的資料也出現問題，會造成的麻煩會變得更大，於是我們就會想要把 Web 與 Database 的機器相互分離。

當用這樣的架構之後，你會發現效率會比之前提高很多，服務彼此不會搶資源了，但是就是要付出 $ 去添購機器摟（當可以用 $ 解決的問題都是小問題）

架構圖：



使用時機

同時上線人數約 20 ~ 40 人左右

人數為 ***KeJyun*** 過去經驗大概估算的人數，沒有經過實際測試僅參考用

當我們的服務能夠尤多台機器去分攤時，機器彼此的處理效率當然就會提高許多，可以乘載的人數也會提高

以飲料店來當例子，1 個服務生可以同時服務 3 位客人（主機同時有 3 個人發出請求），但是若有 2 位服務生就可以同時服務 6 位客人（主機同時有 6 個人發出請求）， 將替客人做點餐結帳、做飲料的工作做分工處理，一為服務生同時可以替 6 位客人做點餐結帳的工作，另一位則可以同時替 6 為客人做飲料，整個服務可以乘載的量就會提高很多， 讓每個人專心做一件事情，效率也會提高很多

> 同時上線非每日上限，同日上線指的是同一時先有多個人同時跟伺服器要資源

需要維持一定的服務可用度

當服務需要提供比較多人去做存取時，我們就必須要提供一定水準的服務可用度，將反應時間保持在 1 ~ 2 秒內的範圍。

資料定期備份

雖然服務還在剛起步的階段，但是資料還是非常重要的公司資產，若只有一台 Database，我們還是需要定期的將資料備份出來（最少 1 天備援一次），避免機器掛掉的機會。

參考資料

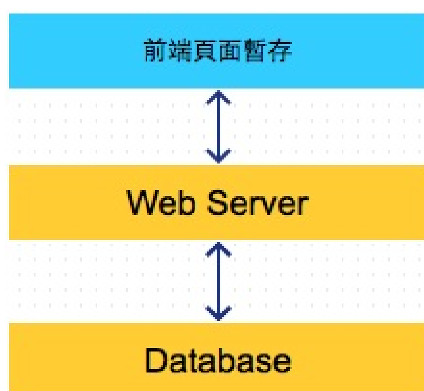
- 一步步構建大型網站架構- 架構設計- | 九街| 白開水的博客

前端靜態頁面暫存

當越來越多人來存取你的應用時，你會發現你服務的 `反應時間` 又開始變慢了，你可以會發現 `Web` 機器效能的使用沒有 `Database` 那麼大，就會發現現在的瓶頸應該是卡在 `Database` 有太多人去進行存取了。

我麼這時候會試著將頁面中很少異動的頁面（大約 1~2 天才會更新的頁面），做成靜態頁面的快取，在撈取資料庫資料時把資料產生成靜態 HTML 檔案，當下次再次的讀取相同資料時，則直接將靜態的 HTML 回傳，減少資料庫的存取，提高存取資料庫的效率（有需要再去進行查詢）。

架構圖：



注意事項

整頁靜態頁不可頻繁的修改

因爲會將整個頁面都做是要給使用者看到的整個結果頁，所以頁面的資料若時常修改的話，則勢必要一直重新產生新的靜態頁面，但這樣就失去了做靜態頁面暫存的意義了

參考資料

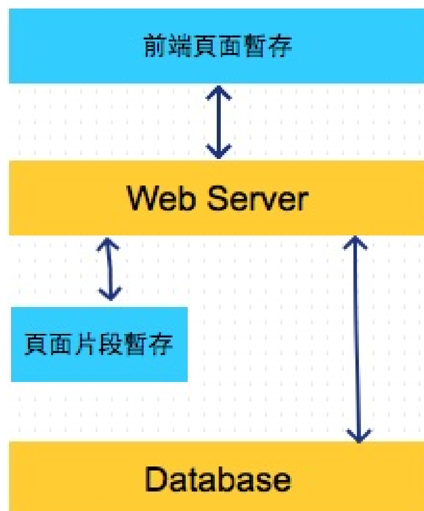
- 一步步構建大型網站架構- 架構設計- | 九街| 白開水的博客

頁面片段暫存

在我們將靜態頁面暫存起來後，減少了很多對資料庫不必要的存取，`回應時間` 提升了了很多。

但我們也會想看看能不能在動態產生的頁面中，對於部分的很少變動的資料也做暫存的處理，這樣的話就可以減少部分動態網頁頁面不必要的存取了。

架構圖：



參考資料

- 一步步構建大型網站架構- 架構設計- | 九街| 白開水的博客

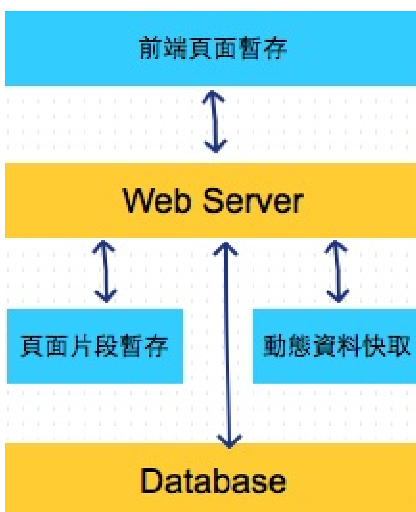
動態資料快取

`產生的靜態頁面暫存` 可以應付整個頁面很少改變的網頁，但是若網頁中有一部分的資料遭到異動，爲了讓使用者取得到最新的資料，我們勢必要清除頁面並重新撈取資料庫的資料去產生新的 `靜態頁面暫存` 。

但對於資料層來說，其實只有 `少部分的資料` 有異動需要重新讀取，若我們需要重新去資料庫撈取 `所有` 頁面需要的資料，再去兜頁面所需要的資訊，勢必是浪費資料庫資源的，還記得我在概念章節中有提到說 `資料庫存取是很昂貴的` ，過多的查詢，導致資料庫的存取效率降低，且資料庫的的連線因爲系統的限制，所以也沒辦法無限制的增加連線數量。

所以我們會想要把一些很少異動的資料去做快取（Cache）預存下來，當下一位使用者來索取相同的資料時，則不去資料庫進行查詢，直接將之前預存的資料丟回給使用者，提高系統反應的時間及資料存取的效率，快取的資料會依照我們設定資料的過期時間，當超過過期時間後，再重複去資料庫撈取資料，看看有沒有異動。

架構圖：



注意事項

快取時間

快取的失效時間通常會依我們的應用去做設定，也因爲系統資源的關係（記憶體大小、硬碟容量大小...等等）的原因，沒有辦法將快取設定爲永久存放，像是做討論區的文章，通常熱門的文章在短時間會被重複的讀取，但熱度可能會隨著時間做遞減，可能大約 3 天的時間文章就會變得越少人存取，當沒有人存取的時候，我們就會希望將快取清除，避免佔住系統資源（記憶體或硬碟容量），所以我們可能會針對討論區的文章做 3 天時間的快取，等超過 3 天

後，快取會自動地將過期的快取資料做清除，除非又有人再次讀取該篇文章，才會再做一次快取（失效時間再次設爲 3 天），一直重複這樣的動作，等到資料不再被存取，就只會保留在資料庫的洪流當中。

手動清除快取

在部落格發表的文章被讀取時，我們會對文章進行快取，但有時文章可能因爲作者「打錯字」或者「需要補充資料」，導致文章需要被異動更新，爲了能夠讓快取有最新被異動過的資料，通常我們會手動的清除這篇文章的快取，這樣就可以確保下一個存取這篇文章的讀者，一定會拿到最新的文章資料。

資料分散式快取

快取的資料通常是用 `key` 及 `value` 的方式去紀錄資料，而我們通常會把資料做片段的快取，像是部落格文章 A 的快取我們通常會存放在 `key` 爲 `blog_post:post_id_A` 的快取資料中，以此類推，文章 B 的快取 `key` 爲 `blog_post:post_id_B`，用這樣的方式對每個不同的小資源去做快取，而不是將所有部落格文章 A 及 B 存放在一整個快取 `key` 爲 `blog_post:all` 中，因爲當使用者異動文章 A 的時候，若我們要刪除快取的資料則只需要刪除文章 A 的快取即可，而不需要把其他不必要刪除的文章快取也一併刪除了，提高快取使用的效率，通常要看自己的應用及使用者存取資料的方式，需要用哪一種方式的快取比較適合，每一種應用都有適合自己的快取方法。

參考資料

- 一步步構建大型網站架構- 架構設計- | 九街| 白開水的博客

增加 Web Server

隨著網站應用使用者增加，我們會發現 Web Server 機器的壓力變得比較高了，這個時候就會開始考慮增加 Web Server 了。增加 Web Server 除了可以降低機器的壓力外，同時也可以在 Web Server 掛掉後，有備援機器可以使用，避免整個服務完全中斷。

在增加 Web Server 之後，會碰到一些問題：

- 如何將流量平均分配到不同的 Web Server

> 通常我們會使用 Apache 的負載平衡方法，或是 LVS 相關的負載平衡方法去分配這些流量

- 如何讓 Web Server 的資料同步

> 像是使用者的 session 在不同機器要如何同步，可能會考慮使用資料庫、cookie 或同步 Session 的機制

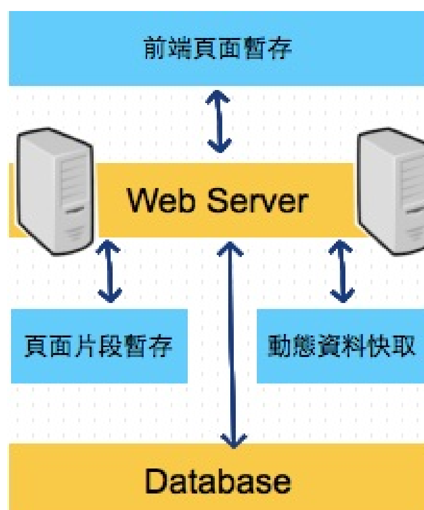
- 快取（cache）資料同步

> 像是快取會員的個人資料，可能會使用快取同步的機制，或者是分散式快取的方式

- 上傳的檔案同步

> 可能會使用檔案共享的方式

架構圖：



參考資料

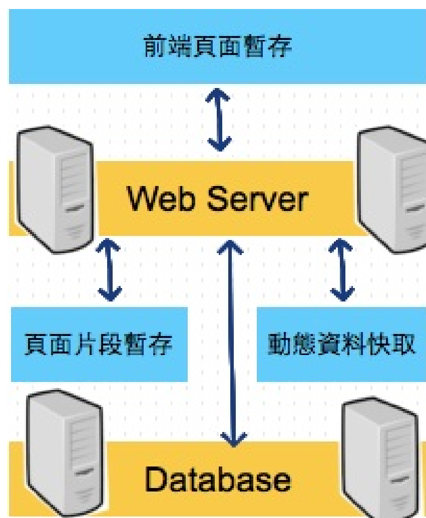
- 一步步構建大型網站架構- 架構設計- | 九街| 白開水的博客

增加 Database Server

在網站成長之後一段時間後，發現資料庫在查詢（SELECT）及資料異動（INSERT、UPDATE、DELETE）之間，因爲查詢處理變得太多，使得互相競爭資料庫的資源非常嚴重，進而導致系統回應時間變得很慢。

這時候我們可能會想要把不同的資料表分散在不同的資料庫當中，分散查詢處理的資源，而會需要更改程式讀取資料庫的邏輯架構。

架構圖：



參考資料

- 一步步構建大型網站架構- 架構設計- | 九街| 白開水的博客

分散式資料

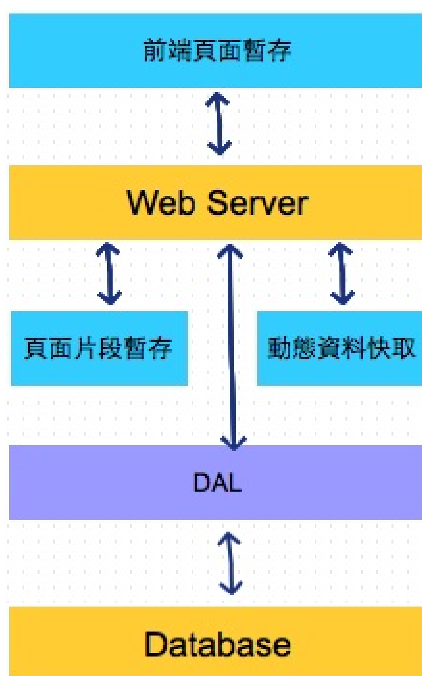
在做資料庫分散之後，若單一資料表的資料太多，我們會想要用分散資料庫的概念，對資料表中的資料進行分散式處理，而會再需要更改程式讀取資料庫及資料表的邏輯架構。

而在這我們就需要規劃資料分散在多個資料庫及資料表的架構，要怎麼去存取，我們通常會設計一個統一的資料庫及資料表資源分散的統一架構，去進行管理資源的存放及存取方式。

像是 Facebook 每天有好幾百萬千萬的文章資訊會被發表出來，我們可能會對各個使用者做文章存取的資源分配，將使用者的資源分配記錄記在統一個資源管理架構中做記錄，紀錄的資訊可能會是 User A 的文章資料存放在 DB_1 及 TABLE_2，而 User B 的文章資料存放在 DB_3 及 TABLE_1，當要存取個別使用者的資料資料，根據資源管理分配的的方式，分別去不同的資料庫及資料表撈取資料。

而當需要使用分散式資料的情況下，可能也會發現因爲資料量過大，導致快取同步的方式出了問題，讓快取不能只存放在本地端而用同步的方式進行分享，可能需要採用分散式快取的方式去快取資料。

架構圖：



參考資料

- 一步步構建大型網站架構- 架構設計- | 九街| 白開水的博客

增加更多的 Web Server

在做完分散式資料庫及分散式資料表之後，又開始看到流量暴增的的時候，發現系統的回應時間又開始變慢了，你可能會看到 Web Server（Apache、Nginx...etc）那邊阻塞了很多的請求，而導致每個存取的使用者需要時間等待而導致回應時間增加了，這個時候我們就可能需要增加更多的 Web Server 去應付這些大量的請求。

在做完這些工作之後，會進到一個像是可以無限擴充的階段，網站流量增加時，就是不斷的增加 Server （$$$） 去應付大量的流量。

在做 Web Server 負載平衡時可能會遇到下列問題：

- 基本的負載平衡（Apache 負載平衡、LVS...etc）無法應付巨量的請求量

如果經費允許的話（都流量暴增了，會不想要投入更多的 $ 去讓自己的事業擴大嗎 XD），可以採購負載平衡的硬體設備

> e.g. F5 Load Balancer、Netsclar Load Balancer...etc

若真的經濟不允許，可以將應用從邏輯上做分類，不同類型的應用分散給不同的負載平衡群集處理。

- 訊息同步的文件分享方法出現瓶頸

這時可以根據不同的網站業務需求去做不同的分散式文件系統

參考資料

- 一步步構建大型網站架構- 架構設計- | 九街| 白開水的博客
- F5 Load Balancer
- Netsclar Load Balancer

Database 讀寫分離

當流量又開始暴增時，會發現原本無限制擴充 Server（$$$）的架構會讓資料庫連線的資源不敷使用。

因爲而在做資料異動的時候，若有要更新範圍（range）的資料，很容易把資料進行鎖定，進而彼此影響到查詢上面的效能。 在 80/20 法則中，大部份 80% 人都是在看文章比較多（讀取資料：SELECT），只有少部分 20% 的人或意見領袖，才會發表文章表示看法（異動資料：INSERT、UPDATE、DELETE），而在做資料異動的時候很有可能會對資料進行鎖定，進而去影響讀取的速度。若是要異動範圍（range）的資料，很容易把資料進行鎖定，更會影響到彼此查詢上面的效能。

這時候我們就會想要把資料庫做 `讀寫分離` ，所以像是 Facebook、部落格之類的媒體，大多會把資料庫做 `讀寫分離` ，使用者做異動的行爲會去主資料庫（Master）去做寫入的動作，然後從資料庫（Slave）在定期的去同步資料庫的內容，而其他大部份的讀者在讀取資料時，都去讀取從資料庫（Slave）的資料，這樣就不會有因爲資料異動而導致資料被鎖定，造成讀取變慢的問題。

而在主從架構中，可以是有很多台從的資料庫（Slave），透過分散式處理可以將不同的連線分配給不同的從資料庫（Slave），讓每一台從的資料庫平均分配差不多的連線量，因爲需要處理的連線減少了，進而讓每個查詢都能夠在短時間都能夠回應查詢結果，提高系統的可用度。

> 資料庫主從架構中， `主（Master）` 資料庫用來做 `寫入資料異動（INSERT、UPDATE、DELETE）` 的動作， `從（Slave）` 資料庫用來做 `讀取（SELECT）` 的動作。

而若有些很少在查詢的資料，像是 `Log 紀錄` 或是在做 `分析的資料` ，這類的資料從放在可靠性較高的資料庫（MySQL、Postgres、Oracle...etc）中，對於我們來說是比較浪費的，因爲他會佔用存取資料庫的資源，這類的資料我們可能可以考慮用一些 NoSQL 的方案去存取這些資料，讓重要的資料存放在可靠性的資料庫就好，提高資料庫的可用性。

參考資料

- 一步步構建大型網站架構- 架構設計- | 九街| 白開水的博客
- 80/20 法則

高可用性的服務架構

透過不斷的增加 Web Server 就可以提高使用者訪問量，但這樣的架構非常的龐大，當要對這樣的架構進行改動時會相當的不方便，可用性變得不高。

而且部署機器和維護也必變得相當麻煩，龐大的應用服務架構要在 N 台機器上進行複製、啓動都需要花很多的時間，當機器出問題時也很難立即找出拿台機器出了問題，更有可能是某個應用服務的程式出現 Bug，導致整個站掛掉都沒辦法使用了。

在優化調校時也比較難操作，因爲部署的機器每一台都需要進襲調校，沒辦法只進行針對性的調校。

餘是爲了解決這樣的問題，就發展出大型的分佈式應用，而這樣的架構可能會遇到許多挑戰：

- 分布式應用需要提供高效能且穩定的架構
- 將龐大的應用拆分出來需要耗費很長的時間，而且需要對業務的整理和系統依賴關係進行控制
- 如何維護（依賴管理、執行狀況管理、錯誤追踪、調校優化、監控和示警等）好這個龐大的分佈式應用。

完成這一步後，系統的架構差不多會到相對穩定的階段，也會使用大量便宜的機器去支撐突如其來的訪問流量

參考資料

- 一步步構建大型網站架構- 架構設計- | 九街| 白開水的博客

常見問題

這裏會解說一些效能相關的問題

壓力測試

在我們寫完程式並上了伺服器後，若我們的網站需要服務很多人，我們可能會想要這個服務能夠服務多少人，通常我們會對伺服器進行壓力測試。

壓力測試的軟體有很多，我比較常用的是 `Apache AB test` ，簡單的了解伺服器在多少連線下，處理請求（Request）的速度及效能，而同樣的機器在不同的程式處理之下，會有不同的效能，所以沒辦法統一的斷定哪一種機器只能服務多少人，若程式寫的好的話，可能同時可以應付 4、500 人同使請求，但也有程式可能因爲效能不佳，僅能服務 4、50 人（伺服器回應時間標準一樣），所以要對自己的應用服務做壓力測試後才知道狀況。

也有很多其他的壓力測試的軟體，只要找出一套適合自己的壓測工具就可以了。

參考資料

- ab - Apache HTTP server benchmarking tool
- The Will Will Web | 使用 ApacheBench 進行網站的壓力測試
- Web server benchmarking - Wikipedia, the free encyclopedia

參考資料

這裏會紀錄一些相關的參考資料，可以去做進一步的學習及查詢

參考書籍

MySQL

- High Performance MySQL, 2nd Edition原文書免費下載

參考文章

概念

- 一步步構建大型網站架構- 架構設計- | 九街| 白開水的博客
- 10+個高流量網站架構的效能提升解決方案
- 100 open source Big Data architecture papers for data professionals.

資料庫

連線池

- Connection pool - Wikipedia, the free encyclopedia

MySQL

- KeJyun學習日誌: 在MySQL中MyISAM與InnoDB資料庫引擎比較

實際案例

- Stack Overflow: The Architecture - 2016 Edition
- 《英雄联盟》玩家聊天服务的持久层演进
- KKBOX Case Study – Amazon Web Services (AWS)

HTTP Cache

- 循序漸進理解 HTTP Cache 機制 | TechBridge 技術共筆部落格

Python

- Python + Django 如何支撑了 7 亿月活用户的 Instagram？
    - Sharding & IDs at Instagram
    - Dismissing Python Garbage Collection at Instagram

參考影片

流量分散式處理

- Load Balancing Stateless vs. Stateful Applications

參考工具

JavaScript

- jsPerf — JavaScript performance playground

Database

連線池

- Database Connection Pool Library | Libzdb